Chapitre 3
Connexion aux bases de données avec Exposed

1. Introduction

Exposed est un framework open source conçu pour s'intégrer à une base de données relationnelle. Entièrement écrit en Kotlin, il bénéficie du soutien de l'entreprise JetBrains à l'origine du langage Kotlin. Exposed est apparu au cours de l'année 2019 en tant qu'alternative plus légère aux frameworks de base de données utilisés dans l'écosystème Java, tels que Hibernate. Ce framework s'appuie sur la syntaxe du langage Kotlin pour simplifier et faciliter l'écriture des requêtes SQL.

Exposed se base sur la librairie JDBC. Cette dernière fournit une couche d'abstraction permettant la connexion aux différents systèmes de gestion de bases de données relationnelles. Chaque base de données relationnelle fournit des *drivers* implémentant la spécification JDBC, ce qui permet de découpler le code de l'application de la gestion de connexion vers une base données spécifique. Cependant, ce découplage se limite à la connexion et à l'exécution des requêtes. Les requêtes, quant à elles, restent propres à chaque système de base de données. En effet, même si tous ces systèmes utilisent le même langage « SQL », chacun d'entre eux implémente un dialecte SQL différent des autres. L'utilisation d'un framework, tel que Exposed ou Hibernate permet de découpler complètement le code de l'application du choix de la base de données relationnelle. Ce choix pourra alors être changé ultérieurement sans affecter le code applicatif ni la logique métier de l'application.

Cette flexibilité est rendue possible grâce au framework qui se charge de générer les requêtes SQL en fonction de la base de données utilisée lors de l'exécution de l'application. Le framework Exposed est compatible avec la plupart des systèmes de gestion de bases données modernes tels que PostgreSQL, SQL Server, MySQL, Oracle et H2.

Pour faciliter son adoption, Exposed permet au développeur d'utiliser deux styles de programmation pour générer les requêtes SQL. Le premier style est un style classique inspiré du framework Hibernate, qui consiste à utiliser le patron de conception DAO (*Data Access Object*). Il repose sur la définition des objets de mapping relationnelle entre le code de l'application et le schéma de la base de données correspondant. Le deuxième style exploite la puissance du langage Kotlin pour offrir un DSL (*Domain Specific Language*) permettant de représenter les différentes requêtes SQL. Cette approche offre davantage de contrôle sur le code SQL généré par le framework. Ce DSL est très similaire à celui des autres frameworks Java tels que jOOQ.

2. Comparaison entre Exposed et Hibernate

Les deux frameworks offrent une méthode fiable pour découpler l'application de la base de données. En étant un langage de la JVM, Kotlin permet d'utiliser facilement tous les frameworks Java existants permettant la gestion de bases de données pour une application. Le choix du framework devrait donc être basé sur les avantages offerts par l'architecture sur laquelle il repose.

2.1 Maintenabilité

La maintenabilité du code de l'application est très importante. Elle implique la nécessité d'avoir un code clair et compréhensible et la capacité à déboguer l'application facilement en cas de problème. La richesse du framework Hibernate, ainsi que sa complexité, peuvent rapidement affecter la lisibilité du code de l'application. Bien que la représentation du schéma de base de données en mapping d'objet relationnel facilite le développement lorsque le schéma est simple, dès que celui-ci comporte plusieurs relations complexes entre les tables, la complexité du code de l'application augmente et le code SQL généré devient moins lisible. Cela rend le débogage plus complexe.

Grâce au paradigme DSL offert par le framework Exposed, le développeur conserve un contrôle complet sur les requêtes générées, ce qui facilite la maintenance de l'application et améliore la clarté du code.

2.2 Maturité

Le framework Hibernate a été développé en 2001. La spécification JPA implémentée par Hibernate a quant à elle été créée en 2006. Il s'agit d'un framework très mature utilisé aujourd'hui dans la grande majorité des applications web ciblant la JVM. Il est toujours activement maintenu par une vaste communauté open source dirigée par l'entreprise Red Hat.

De son côté, Exposed est un framework très récent sorti en 2019. Il a été développé par la même entreprise qui a créé le langage Kotlin. Le framework commence à gagner en popularité au sein de la communauté des développeurs Kotlin. Cependant, il n'a pas encore réussi à atteindre le même niveau de maturité que le framework Hibernate. Jusqu'à la rédaction de cet ouvrage, la première version majeure « 1.0.0 » n'a pas encore été publiée.

2.3 Performance

Grâce à son niveau de maturité, Hibernate a été optimisé au fil des années. Il a réussi à atteindre un bon niveau de performance. Ce niveau de performance demeure néanmoins inférieur à celui atteint en exécutant directement des requêtes SQL via la bibliothèque Java JDBC.

Malgré son manque de maturité, le framework Exposed a réussi à obtenir des performances proches de celles du framework Hibernate.

Généralement, quel que soit le framework choisi, les performances peuvent varier en fonction des cas d'utilisation et de la manière dont il est intégré avec le code de l'application.

3. Installation et configuration

3.1 Ajout des dépendances

Le framework Exposed est disponible sous forme de plusieurs dépendances. Parmi elles se trouvent :

– `exposed-core` : il s'agit du module principal du framework. Il comporte les différentes implémentations des API du framework ainsi que la définition du langage DSL d'écriture des requêtes SQL.
– `exposed-jdbc` : il s'agit du module de gestion des connexions aux bases de données. Il repose sur la bibliothèque Java JDBC.
– `exposed-dao` : il s'agit du module de gestion du mapping objet relationnel. Il repose sur le patron de conception DAO. Ce module est optionnel.

Pour un projet Gradle, ces trois dépendances doivent être ajoutées comme suit :

```
implementation("org.jetbrains.exposed:exposed-core:0.45.0")
implementation("org.jetbrains.exposed:exposed-jdbc:0.45.0")
implementation("org.jetbrains.exposed:exposed-dao:0.45.0")
```

Dans le cadre d'un projet Maven, la configuration de ces dépendances est la suivante :

```
<dependency>
    <groupId>org.jetbrains.exposed</groupId>
    <artifactId>exposed-core</artifactId>
    <version>0.45.0</version>
</dependency>
<dependency>
    <groupId>org.jetbrains.exposed</groupId>
    <artifactId>exposed-jdbc</artifactId>
    <version>0.45.0</version>
</dependency>
<dependency>
    <groupId>org.jetbrains.exposed</groupId>
    <artifactId>exposed-dao</artifactId>
    <version>0.45.0</version>
</dependency>
```

Comme Exposed est basé sur JDBC, l'application doit ajouter la dépendance qui correspond au driver JDBC du système de base de données choisi. Par exemple, pour PostgreSQL, il faut ajouter la dépendance suivante :

– Gradle

```
implementation("org.postgresql:postgresql:42.7.1")
```

– Maven

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.1</version>
</dependency>
```

3.2 Configuration de la connexion

Exposed interagit avec la base de données à travers la classe `org.jetbrains.exposed.sql.Database`. Cette classe est responsable de toutes les configurations relatives à l'établissement de la connexion avec une base de données. La connexion peut être établie en utilisant la méthode `connect` de cette classe. L'exemple ci-dessous illustre la configuration d'une connexion à une base de données PostgreSQL locale :

```
Database.connect("jdbc:postgresql://localhost:5432/
my_db", driver = "org.postgresql.Driver")
```

Le premier paramètre représente la configuration JDBC, le deuxième paramètre spécifie la classe principale du driver JDBC. Ces deux paramètres varient en fonction de la base de données choisie.

La bibliothèque JDBC définit l'interface `java.sql.DataSource`. Cette dernière permet de définir la configuration nécessaire pour établir une connexion à la base de données. Il s'agit du point d'entrée principal pour la création des objets de connexion permettant d'exécuter les requêtes SQL. Chaque driver JDBC fournit une implémentation de cette interface `DataSource`. Exposed prend en charge la création de l'objet `DataSource` correspondant à la base de données utilisée. L'application peut néanmoins choisir de ne pas déléguer la création de cet objet. Dans ce cas, elle en sera responsable. Une fois l'objet créé, il pourra être passé en paramètre à la méthode de connexion `connect` comme suit :

```
Database.connect(dataSource)
```

Il est recommandé d'utiliser cette dernière méthode afin d'avoir un contrôle total sur la gestion des connexions vers la base de données. De plus, cette approche peut être optimisée en utilisant des pools de connexions. Ces pools permettent de réutiliser les connexions pour l'exécution de plusieurs requêtes SQL et offrent donc une meilleure performance. En effet, l'établissement de la connexion à une base de données est une opération très coûteuse pour l'application. Il existe plusieurs outils facilitant la création et la gestion d'un pool de connexions. Parmi eux figure la bibliothèque open source HikariCP. L'utilisation de cette bibliothèque nécessite la dépendance suivante :

– Gradle

```
implementation("com.zaxxer:HikariCP:5.1.0")
```

– Maven

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>5.1.0</version>
</dependency>
```

HikariCP représente la configuration de la connexion par l'objet `com.zaxxer.hikari.HikariConfig` :

```
import com.zaxxer.hikari.HikariConfig

val config = HikariConfig()
```

Cet objet expose plusieurs propriétés permettant de gérer la configuration de la connexion. La première propriété obligatoire concerne l'URL de connexion JDBC :

```
config.jdbcUrl = "jdbc:postgresql://localhost:5432/my_db"
```

La deuxième propriété nécessaire est la classe principale du driver JDBC utilisé :

```
config.driverClassName = "org.postgresql.Driver"
```

Il existe d'autres propriétés optionnelles telles que l'utilisateur et son mot de passe :

```
config.username = "pgUser"
config.password = "secret"
```

Une fois la configuration définie, l'objet `DataSource` de HikariCP peut être créé :

```
import com.zaxxer.hikari.HikariDataSource

val dataSource = HikariDataSource(config)
```

Cet objet `DataSource` pourra ensuite être passé à Exposed afin d'initialiser la connexion entre l'application et la base de données :

```
DataSource.connect(dataSource)
```

3.3 Gestion des transactions

Le framework Exposed requiert que toute opération soit exécutée au sein d'une transaction de manière explicite. Ceci oblige le développeur à bien définir le scope de chaque requête vers la base de données. Une nouvelle transaction est créée en utilisant le bloc `transaction` :

```
import org.jetbrains.exposed.sql.transactions.transaction

transaction {

}
```

Ce bloc est une méthode qui prend une lambda comme paramètre. La lambda sera alors exécutée dans le contexte de la transaction définie par le bloc. Il est possible d'exécuter plusieurs opérations à l'intérieur du même bloc. À la fin de la lambda, la transaction est automatiquement fermée par Exposed en envoyant le signal « commit » à la base de données. Il est également possible de « commit » une transaction à l'intérieur du bloc lorsque l'application a besoin de rendre les changements visibles à d'autres transactions. Pour cela, il suffit d'appeler la méthode `commit` à l'intérieur du bloc :

```
transaction {
  commit()
}
```

Exposed permet également de signaler au système de gestion des données que les changements de la transaction en cours doivent être annulés. Cette opération pourra être réalisée en appelant la méthode `rollback` comme suit :

```
transaction {
  rollback()
}
```

Généralement, l'opération « rollback » est nécessaire lorsqu'une erreur est levée durant le traitement d'une ou de plusieurs requêtes par l'application. Chaque opération « rollback » annule tous les changements réalisés à l'intérieur du bloc de la transaction en cours, s'ils ne sont pas encore appliqués par l'opération « commit ». Il est tout à fait envisageable d'avoir, dans la même transaction, plusieurs appels aux méthodes `commit` ou `rollback`. Par exemple :

```
transaction {
  // operation 1
  commit()
    // operation 2
    rollback()
}
```

Lors de l'exécution de la méthode `rollback`, seuls les changements réalisés par l'« opération 2 » seront annulés.

Par défaut, le bloc `transaction` détecte l'existence d'une transaction en cours avant d'en créer une nouvelle. Dans ce cas, la transaction en cours sera utilisée. Ainsi, il est possible d'avoir des blocs `transaction` imbriqués comme suit :

```
transaction {
    transaction {
        transaction {

            }

        transaction {

        }
    }
}
```

Dans le cas précédent, le bloc `transaction` parent est responsable de la gestion de la transaction. Celle-ci ne sera fermée que lorsque toutes les opérations du premier bloc seront terminées. Toutes les transactions enfants du bloc transaction parent n'entraînent pas la création d'une nouvelle transaction. Le commit des opérations effectuées dans une transaction enfant aura lieu à la fin de la transaction parent si aucun commit explicite n'est utilisé.

4. Les bases du framework Exposed

4.1 Représentation des tables

4.1.1 Déclarer les tables

Exposed représente les tables d'une base de données à travers la classe `org.jetbrains.exposed.sql.Table`. Chaque instance de cette dernière est associée à une table dans le schéma de la base de données de l'application. Exposed repose sur le patron de conception singleton et utilise la syntaxe Kotlin pour la modélisation des différentes tables.

Prenons l'exemple de la classe métier suivante :

```
data class Team(id: Int, name: String)
```